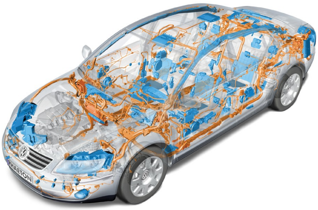
Clock Refinement in Imperative Synchronous Languages

Mike Gemünde

October 18th, 2013

Model-Based Design and Models of Computation



- (parallel, distributed) models of computation (MoC)
- abstract special properties, focus on relevant attributes (e.g. communication)
- e.g. discrete event, data-flow process networks, **synchronous model**

Synchronous Model of Computation

Ideal World (Development)

- produce the outputs synchronously with the inputs
- abstract from delay of computation (micro steps)
- (logical) time is consumed between reactions (macro steps)
- compose very well
- focus on logic of reaction

Real World (Execution)

- challenges compilers
- requirements of application must be met

Various Languages

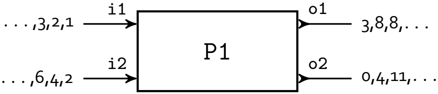
Data-Flow: Lustre, Signal Control-Flow: Esterel, **Quartz**, Statecharts

Quartz Example P1

```
module P1 (nat ?i1,?i2,o1,o2)
{
  nat x;
  loop {
    o1 = i1 + i2;
    x  = i1;
    pause;
    o1 = o2 + i1 + x;
    o2 = i2;
    x  = 2;
    pause;
    if (i1 > 4)
      o1 = i1;
    o2 = i1 + o1;
    pause;
  }
}
```

- **pause** marks end of a (macro) step
- inputs: i1, i2 outputs: o1, o2
  local variable: x
- new inputs/outputs in each step
- execution follows data dependencies

	1	2	3	4	5
i1	1	2	3	4	5
i2	2	4	6	8	0
x	1	2	2	4	2
o1	3	8	8	12	7
o2	0	4	11	11	0



Quartz Statements

- assignments: `x`=α, **`next`**`(x)`=α
- end of step: **`pause`**
- conditional execution: **`if`**`(`γ`)... ` **`else`** `...`
- loops: **`while`**`(`γ`){ ... }`, **`loop`**`{ ... }`
- waiting: **`await`**`(`γ`)`
  - also time consuming
- abortion: **`abort`** `...` **`when`**`(`γ`)`
  - various variants
  - aborts execution when condition γ holds
- suspension: **`suspend`** `...` **`when`**`(`γ`)`
  - various variants
  - suspens execution when condition γ holds
- **concurrent execution:** `{ ... } || { ... }`
- ...

Contribution

Extension to define Substeps for Imperative Synchronous Languages

- introduce temporal refinement
- not limited to structural abstraction
- more possibilities for re-use
- stay in the same model
- showed for QUARTZ

Outline

- Introduction of the Extension
- Definition of Semantics
- Compilation to Intermediate Format
- Synthesis from Intermediate Format & Evaluation

Outline

- Introduction of the Extension
- Definition of Semantics
- Compilation to Intermediate Format
- Synthesis from Intermediate Format & Evaluation

Example (Greatest Common Divisor)

```
module GCD(nat ?a,?b,!gcd)
{

    nat x = a; y = b;
    while(x > 0) {
      if(x >= y)
        next(x) = x-y;
      else
        next(y) = y-x;
      pause;
    }
    gcd = y;
    pause;

}
```

- Euclidean Algorithm in Quartz
- dynamic number of steps needed for computation
- result is not directly available
- calling module must take care of time consumption

	1	2	3	4	5	6
a	**7**	7	7	7	7	7
b	**3**	3	3	3	3	3
x	7	4	1	1	1	0
y	3	3	3	2	1	1
gcd	0	0	0	0	0	**1**

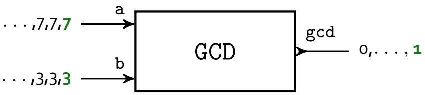
Example (Greatest Common Divisior)

```
module GCD(nat ?a,?b,!gcd)
{

    nat x = a; y = b;
    while(x > 0) {
      if(x >= y)
        next(x) = x-y;
      else
        next(y) = y-x;
      pause;
    }
    gcd = y;
    pause;

}
```

- Euclidean Algorithm in Quartz
- dynamic number of steps needed for computation
- result is not directly available
- calling module must take care of time consumption



Example (GCD) - Idea of Refined Clocks

```
module GCD(nat ?a,?b,!gcd)
{
  clock(C1) {
    nat x = a; y = b;
    while(x > 0) {
      if(x >= y)
        next(x) = x-y;
      else
        next(y) = y-x;
      pause(C1);
    }
    gcd = y;
    pause;
  }
}
```

- add declaration of clock C1
- C1 is not visible to outside
- calling module just **sees** one step as whole computation

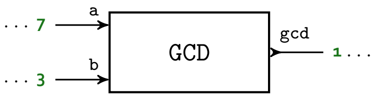
C0	1	—	—	—	—	—
C1	1	2	3	4	5	6
a	**7**	—	—	—	—	—
b	**3**	—	—	—	—	—
x	7	4	1	1	1	0
y	3	3	3	2	1	1
gcd	—	—	—	—	—	**1**

- C0 is considered as module clock

Example (GCD) - Idea of Refined Clocks

```
module GCD(nat ?a,?b,!gcd)
{
  clock(C1) {
    nat x = a; y = b;
    while(x > 0) {
      if(x >= y)
        next(x) = x-y;
      else
        next(y) = y-x;
      pause(C1);
    }
    gcd = y;
    pause;
  }
}
```

- add declaration of clock C1
- C1 is not visible to outside
- calling module just **sees** one step as whole computation

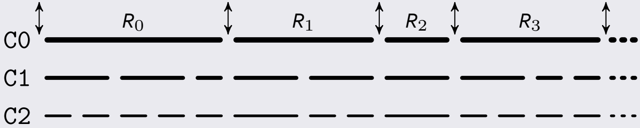


- results available in same step
- **same** language used for computation

Refined Clocks

- not only structural abstraction, but also of timing behavior
- local acceleration of steps (logically)
- steps are divided by sub-steps of lower clocks
- variables of lower clocks can change more often

Refined Clocks/Steps



Synchronization on Parallel Threads

```
module parallel1(...)
{
      pause;  <---||--> pause;

      pause;  <---||--> pause;
}
```

- parallel threads synchronize on **pause** statements
- synchronous composition
- assignments in between are executed in the same step

Synchronization on Parallel Threads

```
module parallel1(...)
{
  clock(C1) {        ||  clock(C2) {
    pause(C1);       ||
    pause; <-------- || --> pause;
    pause(C1);       ||       pause(C2);
    pause(C1);       ||
    pause; <-------- || --> pause;
                     ||       pause(C2);
  }                  ||  }
}
```

- parallel threads synchronize on **pause** statements of same clock
- synchronization on C1 is not possible, because C1 is just visible in one thread
- between two **pause** statements, arbitrarily many steps related to lower clocks can be done (e.g. GCD computation)
- execution of substeps until synchronization point on next common **pause**

More Synchronization on Parallel Threads

module parallel2(...)

```
{
  clock(C1) {
    pause(C1);        ||
    pause;    <- - -  || - -> pause;
    pause(C1);      <-|| ->   pause(C1);
    pause(C1);        ||
    pause;    <- - -  || - -> pause;
                      ||      pause(C1);
  }                   ||    }
}
```

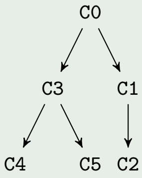
- parallel threads synchronize on common clocks
- synchronization is possible on C0 and C1
- if one thread already reached the **pause** statement of a higher clock, it waits for the other one
- **consequence:** the same step

Clock Tree is Determined by Declarations

module clocktree(...)

```
{
  clock(C1) {
    clock(C2) {
      ...
    }
  }
  clock(C3) {
    clock(C4) {
      ...
    }
    ||
    clock(C5) {
      ...
    }
  }
}
```

$\Rightarrow$



- declaration determines scope
- clock tree can be directly derived from syntax
- C0 is considered as module clock
- again: lower clocks are **faster**

Outline

- Introduction of the Extension
- Definition of Semantics
- Compilation to Intermediate Format
- Synthesis from Intermediate Format & Evaluation

(Traditional) SOS Rules

Plotkin's Approach

- define behavior of programs
- each statement updates **store** and/or **residual statement**
- behavior is completely defined by store
- store is for each statement defined by previous statements
- statements influence only following statements

SOS Rules (Plotkin)

$$\frac{\langle e, \sigma \rangle \rightarrow^{\bullet} \langle m, \sigma \rangle}{\langle \mathtt{x} \; := e, \sigma \rangle \rightarrow \underbrace{\sigma[m/\mathtt{x}]}_{\text{update store}}}$$

Assignment

$$\frac{\langle b, \sigma \rangle \rightarrow^{\bullet} \langle \mathrm{true}, \sigma \rangle}{\langle \mathbf{if}\; b \;\mathbf{then}\; c \;\mathbf{else}\; d, \sigma \rangle \rightarrow \langle c, \sigma \rangle}$$

select if-branch

Conditional

SOS Rules for Quartz

- value of variable is constant for whole step
- all assignments are executed synchronously
- Default Reaction
  - define value of variable if it is not set
  - different for memorized and event variables

$\rightsquigarrow$ divide step into two stages
  - **Reaction Rules**
    - determine environment $\mathcal{E}$ iteratively
    - model **unknown** values with: $\perp$
  - **Transition Rules**
    - step transition

Example

```
{
  if(y > 2)
    x = 1;
} || {
  y = 1;
  z = x;
}
pause;
...
```

$\mathcal{E}$	1	2	3	4
x	$\perp$	$\perp$	0	0
y	$\perp$	1	1	1
z	$\perp$	$\perp$	$\perp$	0

SOS Rules for Quartz

$\langle \mathcal{E}, \hbar, \mathcal{S} \rangle \hookrightarrow_{\mathcal{Q}} \langle \hbar', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, t_{\text{can}}, t_{\text{must}} \rangle$

Reaction Rules

$\langle \mathcal{E}, \hbar, \mathcal{S} \rangle \twoheadrightarrow_{\mathcal{Q}} \langle \hbar', \mathcal{S}', \mathcal{A}^{\text{nxt}}, t \rangle$

Transition Rules

Semantics for the Extension: Default Reaction

Example

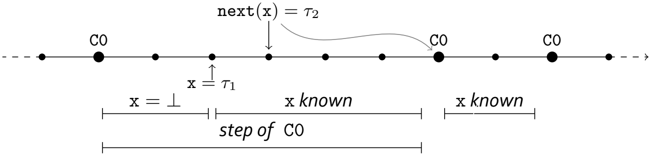
```
int x, y;
x = 3;
clock(C1) {
  int z;
  pause;
  z = 1;
  pause(C1);
  z = 2;
  pause(C1);
  if(z > 3)
    x = τ1;
  pause(C1);
  next(x) = τ2;
  ...
  pause;
}
```

- value of x, y constant for whole step
- value of z can change every substep

⇝ only some of the variables are set to $\perp$ for $\mathcal{E}$

- x is maybe assigned in 3rd substep
- Is x **known** when it is not assigned in the 3rd step?

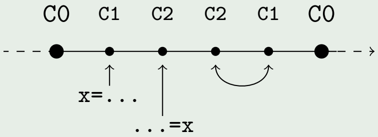
⇝ default reaction needs to ensure that it is also not assigned later in this step



Semantics for the Extension: Choose Clock

```
module P7(nat x)
{
  clock(C1) {
    pause;
    pause(C1);
    x = ...;
    pause(C1);
    pause;
  }
  ||
  clock(C2) {
    pause;
    pause(C2);
    ... = x;
    pause(C2);
    pause;
  }
}
```

Execution Trace



- one value for x for a (module) step
- 2nd substep C2 needs value of x
- step of C1 must be executed first
- then, order is independent since no other dependencies exist
- generally:
  - scheduling order can depend on values
  - each proper scheduling lead to the same result

Summary Semantics

- interpreter and SOS rules of Quartz have been extended
- different approaches for SOS rules have been considered
- this final version explicitly covers two major aspects
  - choose the clock of the next step
  - when is the default reaction triggered
- `DoDefault` not needed for synthesis
- scheduling independence obtained by some restrictions

```
function Instant(ℰ   ℰ   ℰ , C, S)
begin ChooseClock(C)
  c :=
  if c = C0 then ℰin := ReadInputs() else ℰin := ℰ⊥

  ℰprv := (ℰprv)/c⋠ ∪̇ (ℰcur)/c⪯
  ℰcur := (ℰcur)/c⋠ ∪̇ (ℰnxt)/c⪯ ∪̇ ℰin
  ℰnxt := (ℰnxt)/c⋠
  ħinit := {(x,0) | x ∈ 𝒱}

  do # fixpoint iteration
    ℰold := ℰcur
    ⟨ħnew, 𝒜can, 𝒜must, 𝒞can, 𝒞must⟩ :↩c ⟨ℰ, ħinit, , 𝒮⟩

    foreach x ∈ 𝒱loc ∪ 𝒱out do
      ℋ := {ħ(x) | (x = τ, ħ) ∈ 𝒜can}
      foreach i in 0 .. ħ(x) do
        if i ∉ ℋ ∧ ℰcur^ħ
          if i ≠ ħ(x)  DoDefault(x)
            ℰcur := d
      end
    end

    foreach (x = τ, ħ) ∈ 𝒜must do
      ℰcur := [ℰcur]^{[[τ]]^ħ_ℰcur}_(x,ħ)
    end
  while(ℰold ≠ ℰcur)

  ⟨𝒮', ħnew, 𝒜, 𝒞⟩ :↞c ⟨ℰ, ħinit, C𝒮, 𝒮⟩

  𝒞 := 𝒞 ∪ C0
  if ∃x ∈ 𝒱. (∄c ∈ C. c ≺ clock(x)) ∧ ∃0 ≤ i ≤ ħnew(x). ℰcur^i(x) ∈ {⊥, ⊤}
    then Fail()

  foreach (next(x) = τ, ħ) ∈ 𝒜 do ℰnxt := [ℰnxt]^{[[τ]]^ħ_ℰcur}_(x,ħ) end
  ℰcur := {(x, [ℰcur^{ħnew(x)}(x)]) | x ∈ 𝒱}
  ℰnxt := {(x, [ℰnxt^{ħnew(x)}(x)]) | x ∈ 𝒱}

  # write outputs
  if C = {C0} then WriteOutputs(ℰcur)
  return (ℰprv, ℰcur, ℰnxt, 𝒞, 𝒮')
end
```

Outline

- Introduction of the Extension
- Definition of Semantics
- Compilation to Intermediate Format
- Synthesis from Intermediate Format & Evaluation

Compilation of Quartz

- Quartz compiler translates programs to guarded actions (AIF)

Guarded Actions

$$\gamma \Rightarrow \quad \mathtt{x} = \tau \qquad \text{(Immediate Action)}$$
$$\gamma \Rightarrow \mathtt{next}(\mathtt{x}) = \tau \qquad \text{(Delayed Action)}$$

- keep **synchronous** semantics
- abstract from complex control flow
- action is evaluated in an instant when its guard is true
- immediate assignment takes place in **current instant**
- delayed assignment transfers value to **next instant**

Compilation of Quartz Example

```
module P1 (nat ?i1,?i2,o1,o2)
{
  nat x;
  loop {
    o1 = i1 + i2;
    x  = i1;
    l1: pause;
    o1 = o2 + i1 + x;
    o2 = i2;
    x  = 2;
    l2: pause;
    if (i1 > 4)
      o1 = i1;
    o2 = i1 + o1;
    l3: pause;
  }
}
```

Data Flow

$$
\begin{aligned}
\mathtt{st} &\Rightarrow \mathtt{o1} = \mathtt{i1} + \mathtt{i2} \\
\mathtt{st} &\Rightarrow \mathtt{x} = \mathtt{i2} \\
\mathtt{l1} &\Rightarrow \mathtt{o1} = \mathtt{x} + \mathtt{i1} + \mathtt{o2} \\
\mathtt{l1} &\Rightarrow \mathtt{o2} = \mathtt{i2} \\
\mathtt{l1} &\Rightarrow \mathtt{x} = 2 \\
\mathtt{l2} \wedge \mathtt{i1} > 4 &\Rightarrow \mathtt{o1} = \mathtt{i1} \\
\mathtt{l2} &\Rightarrow \mathtt{o2} = \mathtt{i1} + \mathtt{o1} \\
\mathtt{l3} &\Rightarrow \mathtt{o1} = \mathtt{i1} + \mathtt{i2} \\
\mathtt{l3} &\Rightarrow \mathtt{x} = \mathtt{i2}
\end{aligned}
$$

Control Flow

$$
\begin{aligned}
\mathtt{st} &\Rightarrow \mathtt{next}(\mathtt{l1}) = \mathsf{true} \\
\mathtt{l1} &\Rightarrow \mathtt{next}(\mathtt{l2}) = \mathsf{true} \\
\mathtt{l2} &\Rightarrow \mathtt{next}(\mathtt{l3}) = \mathsf{true} \\
\mathtt{l3} &\Rightarrow \mathtt{next}(\mathtt{l1}) = \mathsf{true}
\end{aligned}
$$

Compilation of Example for the Extension

module P (...)

```
l0: pause;
clock(C1) {

    clock(C2) {
      l1: pause(C2);
      y = true;
      l2: pause(C1);
      x = true;
      if(y)
        l3: pause(C2);
      z = true;
      l4: pause;
      y = false;
    }

}
l5: pause;
```

Data Flow

$$\begin{aligned}
C2 \wedge l1 &\Rightarrow y = \text{true}\\
C1 \wedge l2 &\Rightarrow x = \text{true}\\
C2 \wedge l3 &\Rightarrow z = \text{true}\\
C1 \wedge l2 \wedge \neg y &\Rightarrow z = \text{true}\\
C0 \wedge l4 &\Rightarrow y = \text{true}
\end{aligned}$$

Control Flow

$$\begin{aligned}
C0 \wedge st &\Rightarrow \text{next}(l0) = \text{true}\\
C0 \wedge l0 &\Rightarrow \text{next}(l1) = \text{true}\\
C2 \wedge l1 &\Rightarrow \text{next}(l2) = \text{true}\\
C1 \wedge l2 \wedge y &\Rightarrow \text{next}(l3) = \text{true}\\
C1 \wedge l2 \wedge \neg y &\Rightarrow \text{next}(l4) = \text{true}\\
C2 \wedge l3 &\Rightarrow \text{next}(l4) = \text{true}\\
C0 \wedge l4 &\Rightarrow \text{next}(l5) = \text{true}
\end{aligned}$$

Compilation of Example with Abort

```
module P (...)
l0: pause;
clock(C1) {
  abort {
    clock(C2) {
      l1: pause(C2);
      y = true;
      l2: pause(C1);
      x = true;
      if(y)
        l3: pause(C2);
      z = true;
      l4: pause;
      y = false;
    }
  } when(σ);
}
l5: pause;
```

Data Flow

$$\begin{aligned}
\texttt{C2} \wedge \texttt{l1} &\Rightarrow \texttt{y} = \text{true}\\
\neg\sigma \wedge \texttt{C1} \wedge \texttt{l2} &\Rightarrow \texttt{x} = \text{true}\\
\texttt{C2} \wedge \texttt{l3} &\Rightarrow \texttt{z} = \text{true}\\
\texttt{C1} \wedge \texttt{l2} \wedge \neg\texttt{y} &\Rightarrow \texttt{z} = \text{true}\\
\neg\sigma \wedge \texttt{C0} \wedge \texttt{l4} &\Rightarrow \texttt{y} = \text{true}
\end{aligned}$$

Control Flow

$$\begin{aligned}
\texttt{C0} \wedge \text{st} &\Rightarrow \texttt{next}(\texttt{l0}) = \text{true}\\
\texttt{C0} \wedge \texttt{l0} &\Rightarrow \texttt{next}(\texttt{l1}) = \text{true}\\
\texttt{C2} \wedge \texttt{l1} &\Rightarrow \texttt{next}(\texttt{l2}) = \text{true}\\
\neg\sigma \wedge \texttt{C1} \wedge \texttt{l2} \wedge \texttt{y} &\Rightarrow \texttt{next}(\texttt{l3}) = \text{true}\\
\neg\sigma \wedge \texttt{C1} \wedge \texttt{l2} \wedge \neg\texttt{y} &\Rightarrow \texttt{next}(\texttt{l4}) = \text{true}\\
\texttt{C2} \wedge \texttt{l3} &\Rightarrow \texttt{next}(\texttt{l4}) = \text{true}\\
\neg\sigma \wedge \texttt{C0} \wedge \texttt{l4} &\Rightarrow \texttt{next}(\texttt{l5}) = \text{true}
\end{aligned}$$

$$\begin{aligned}
\sigma \wedge \texttt{C1} \wedge \texttt{l2} &\Rightarrow \texttt{next}(\texttt{l5}) = \text{true}\\
\sigma \wedge \texttt{C0} \wedge \texttt{l4} &\Rightarrow \texttt{next}(\texttt{l5}) = \text{true}
\end{aligned}$$

Outline

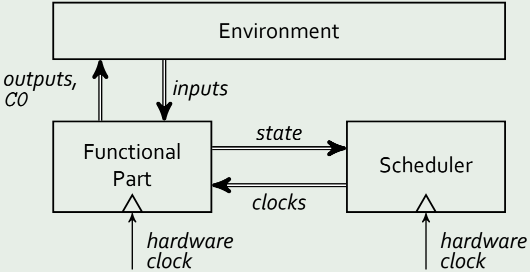
- Introduction of the Extension
- Definition of Semantics
- Compilation to Intermediate Format
- Synthesis from Intermediate Format & Evaluation

Hardware Synthesis

- separate functional part and scheduler
- each variable is translated separately
- scheduler triggers clocks (`ChooseClock`)
- hardware clock is provided from outside
- `C0` is also an output

⇝ inform environment about finished step

Synthesis



Determining a Scheduler

- clocks are not allowed to be arbitrarily triggered
- scheduler needs to respect original semantics
  - internal state (control flow)
  - clock tree (relation of the clocks)
  - data dependencies
- semantics allows re-oder of independent substeps

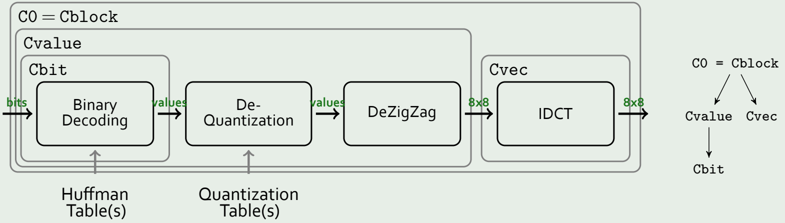
⇝ scheduler can also execute the substeps together

- restrictions on original model makes scheduler straightforward
  - e.g. no immediate assignments between unrelated clock levels
  - no dependencies between lower clocks
  - scheduling restrictions are
    - internal state (control flow)
    - clock tree (relation of the clocks)
    - ~~data dependencies~~

  ⇝ c.f. oversampling in Signal (one tick is required for data exchange)
  - the following JPEG example will show that this is feasible

JPEG Decoding

JPEG Decoding



- Decoder: 0 to 28 Bits per value
- DeZigZag: 64 values per MCU
- IDCT: transform 8x8 matrix
- per (macro) step, one MCU is procuded
- refined clocks abstract data rates
- substeps hide IDCT computation
- I/O only possible with C0, input must be cached

Experimental Results

Example		Quartz		Equations		Circuit		
		# Clocks	# LoC	# Reg.	# Wire	# Reg.	# LUTs	Delay
JPEG	single	1	$\sim 1K$	$307 + 31$	180	$9,132$	$9,049$	20.2ns
	ext.	6		$374 + 29$	191	$11,208$	$11,812$	26.0ns
IDCT2	single (1)	1	~ 350	$130 + 4$	184	$3,995$	$4,047$	25.7ns
	single (2)	1		$148 + 18$	144	$4,973$	$6,780$	11.4ns
	ext. (1)	2		$130 + 4$	188	$4,018$	$4,052$	25.9ns
	ext. (2)	4		$150 + 18$	152	$4,606$	$6,233$	12.9ns
GCD	single	1	15	$3 + 2$	3	33	104	3.9ns
	ext.	2	17	$3 + 2$	7	33	78	3.7ns
TRACE	single	1	~ 100	$10 + 11$	26	57	121	5.8ns
	ext.	3		$17 + 14$	32	76	180	6.5ns

Summary

- extension of imperative synchronous languages with substeps
- different way of thinking for programmers
- describe things in a different way (not more expressive)
- new challenges to define semantics, compiler and synthesis tools

Summary

- extension of imperative synchronous languages with substeps
- different way of thinking for programmers
- describe things in a different way (not more expressive)
- new challenges to define semantics, compiler and synthesis tools

Thank You for your Attention